Full-Disk Encryption Systems: Architecture, Attacks, and AEGIS-Lock: A Threshold-Gated Protocol for Stronger Physical-Access Guarantees

Nico Mangiafico Justin Zhang Kevin Zhao

Abstract

Before getting into our paper, we describe what each team member contributed and focused on. In this paper, Nico worked on the background of existing systems, ideas surrouding our contributions below of key rewrapping, sector authentication, nonces, and measured boot. Justin researched the various attacks surfaces, analyzed the threat model, brainstormed the secret sharing contribution ideas, and research cross-epoch replay prevention. Justin and Nico also collaborated on putting these ideas together into a formal, cohesive model. Kevin worked on implementing the prototype and gathering simulation data, putting them into graphs and visualization, including many of the diagrams in this paper. For writing the paper, we mostly stuck to writing about our own sections, but also collectively wrote certain sections.

Full-disk encryption (FDE) is the prevailing control used to ensure data at rest remains secure on lost or stolen hardware. Products like BitLocker, FileVault 2, LUKS2, and VeraCrypt offer excellent confidentiality in a system that is powered down and properly hardened, but the realistic security margins are often smaller in the face of physical access to the boot process, firmware, TPM bus, or memory. Notably, XTS-AES (the most common mode for FDE) is a purely-encrypting mode, offering data and origin confidentiality without authentication, so a single-bit change in memory can go undetected; the TPM-only unlock option offers key protection, but not full key confidentiality in a single-bus TPM; while in-memory key storage is still vulnerable to cold boot attacks and DMA attacks.

We compare the architecture and security properties of four FDE solutions, and analyze their vulnerability to known physical-access attacks. We then propose AEGIS-Lock, a threshold-gated FDE solution using secret-sharing-based commit-verified shares, epoch-based key wrapping, AEAD sector encryption using deterministic sector-write nonces, TPM-first boot ordering, replay detection across epochs. We develop a Python prototype of the system and evaluate each of the main components, including an Argon2id key derivation of a passphrase into key shares. In the included benchmark run, we evaluate the ordered boot process as taking 0.46,ms in the total path, and the recommended (3,5) 256-bit reconstruction phase as taking on average 101,μs. The MK-driven epoch rotation average is 220,μs. AEAD with a deterministic nonce takes an additional 16 bytes of authentication tag storage per 4,KiB sector (0.39% of space), and this assumes a sector-write epoch or counter is available through trusted metadata. Using the configuration tested here, a 64,MB Argon2id configuration costs 60-139,ms in key derivation, and a 256,MB 2-pass configuration takes 350,ms. AEGIS-Lock is not intended to be a replacement for any FDE solution, but is rather a demonstration that more secure FDE can be designed in a layered architecture, without incurring a whole-disk re-encryption on every key wrap rotation.

1 Introduction

Full-disk encryption (FDE) is arguably the single most widely-implemented mechanism for protecting sensitive data at rest. In general, its primary goal is to guarantee that an adversary faced with a lost, stolen, or disposed system is faced with a computationally expensive problem rather than open filesystem access. The latter is a more realistic risk

in many cases, given the volume and variety of sensitive information that is stored on any hard drive, from passwords, browser history, SSH certificates, or cached documents, to health records or corporate credentials. FDE is currently widely adopted across major platforms: Windows systems ship with BitLocker, macOS ships with FileVault2, Linux systems often default to using LUKS2, and VeraCrypt has emerged as the community-maintained cross-platform successor to TrueCrypt. While the details of key hierarchy, recovery process, hardware-binding, and user experience differ among them, the basic structure is the same: a high-entropy volume key encrypts the disk sectors, and this volume key is only unwrapped/released if some credential is provided, the hardware is verified, a recovery secret is provided, or a combination thereof. That said, a powered-off device in the untampered state (i.e., the boot-chain and unlock process are intact) is really the only state for which the FDE security guarantee holds. Once the adversary is able to re-power-on the device, perform modifications in the early boot stages, connect devices capable of DMA, cold boot the memory (or the memory is already in a state conducive to cold-boot attacks), or observe a TPM bus (external to the host), the security boundary becomes far more complicated. There are three main issues that contribute to the above-mentioned security gap. XTS-AES does not perform authentication as a goal is disk confidentiality [18]. Key(s) are needed in memory, which could be the targets of live memory attacks if the system doesn't use specialized memory protection. Finally, hardware-assisted unlocking improves the security and user experience, however it might create new trust assumptions, e.g., key material leaves a trusted discrete TPM device and gets observable over a TPM bus. We investigate this gap between the security ideal and the deployed systems in this paper. Sec.2 explains the key organization and unlocking process of BitLocker, FileVault 2, LUKS2, and VeraCrypt, Sec.3 specifies the threat model, Sec.4 describes the physical-access attacks that are the most relevant to the FDE security model. In Sec.5, we propose a novel threshold-gated locking protocol called AEGIS-Lock that reduces single points of compromise while introducing the ability to authenticate disk sectors. Sec.6 performs the security analysis of AEGIS-Lock (as well as LUKS2), while Sec.7 discusses and describes which attacks are mitigated and partially mitigated, as well as which attacks are outside the security of FDE.

2 Background: How FDE Systems Work

FDE systems employ layered key architecture in their design. A volume encryption key (VEK) of high entropy encrypts the disk sectors, and this VEK is not derived from the user's passphrase; rather it is protected by the Key Encryption Key(s) (KEK) that are derived from passphrases, hardware secrets, TPM state, or some recovery key or enterprise escrow system. This allows for the system to be able to change the user's password or provide recovery without having to re-encrypt the entire disk.

2.1 BitLocker

The BitLocker volume key architecture uses a hierarchy. The Full Volume Encryption Key (FVEK) encrypts sectors in XTS-AES-128 or XTS-AES-256. The FVEK is then protected by a 256-bit Volume Master Key (VMK), which is protected by one or more key protectors that can be e.g., TPM-only, TPM+PIN, TPM+USB, 48-digit recovery password. In TPM-only mode, the VMK is sealed to the PCRs. A standard configuration for UEFI Secure Boot will use PCR7 and PCR11 with BitLocker. In the absence of PCR7 binding, Windows can alternatively use a profile like PCR0, 2, 4 and 11 [22, 23]. BitLocker is fairly robust to many boot-chain modifications in such scenarios, but auto-unlocking will then depend on the correctness of the platform measurements. With a discrete TPM on LPC or SPI, the unsealed secret can be seen on the bus, unless additional factors are required from the user.

2.2 FileVault 2

FileVault is integrated with Apple's Secure Enclave in Macs with Apple silicon and the T2 chip. Apple says the file system handles encryption key management, and keys are not made directly available to the CPU. The disk is partitioned and contains a volume that is encrypted with a volume encryption

key that is in turn encrypted with a KEK. This KEK is bound to the user password, hardware UID and Secure Enclave, and can not be used to decrypt the volume without these conditions all being met for FileVault to work [24]. FileVault is more of a hardware integration solution; it tightly couples disk encryption with the hardware security of the machine. Attacks would have to happen against the physical device, for example brute force, and erasing keys would effectively destroy the volume. FileVault is therefore not necessarily the strength of the solution, but rather the key management solution and the tight integration with Apple's boot-security model.

2.3 LUKS2

LUKS2 has become the primary disk-encryption container format for Linux. It keeps redundant, machine-readable header information with JSON and provides support for multiple keyslots. Each slot can contain a different key but unlock the same key and encrypt/decrypt the same data. LUKS2 also implements memory-hard password-based key derivation for better resistance to offline guessing, e.g. support for Argon2 variants. The password parameters of this mechanism (PBKDFs) can be specified in the header [20]. The LUKS2 password storage is therefore significantly harder to crack by brute force since parameters such as memory usage, computation time and parallelism can be configured and tuned per keyslot.

Linux also provides authenticated encryption paths for disk encryption. dm-crypt, the default disk encryption device in Linux, can be used in combination with dm-integrity. The latter is described as an integrity protection target for dm-crypt devices which, in the event of any modification to the encrypted device, will result in a failure to access the plaintext data [21]. A more detailed specification can be found in the source code repository. Many standard use-cases for LUKS disk encryption do not currently include the authenticated storage path in their deployments; for example, LUKS1 uses XTS without integrity protection. Enabling integrity protection requires additional data in the header and volume, so there are considerations for disk size and performance.

2.4 VeraCrypt

VeraCrypt is a full-disk and file-system encryption application that provides an easy way to perform full-disk encryption for any volume or block device. It supports many ciphers such as AES, Serpent, Twofish and cascaded combinations of them, in order to prevent single cipher attacks or algorithm weaknesses from rendering the volume vulnerable. VeraCrypt uses the PBKDF2-HMAC key derivations, and has a parameter PIM (Personal Iterations Multiplier) to determine how computationally expensive the key derivation will be. According to the official documentation, the PBKDF2 HMAC default PIM value used by VeraCrypt is 200,000 iterations, with SHA-256, BLAKE2s-256 or Streebog for the system, and 500,000 non-system volume iterations for most encryption algorithms [25]. However, PIM is optional for volumes and system volumes created using earlier versions of the software. The PBKDF2 iterations can therefore be easily modified during volume creation to adjust the time required to derive a key from the user's passphrase. VeraCrypt doesn't rely heavily on TCM. Its use is intended for systems that are already vulnerable and for users who are concerned about portability. The absence of TCM reliance allows for a wider variety of applications and use cases that don't require TPM but also removes the TPM's protection from TPM-specific attack scenarios. This places more emphasis on pre-authentication and user-provided authentication, requiring the user to ensure they are running the boot process on their own machine, rather than the system itself.

2.5 Comparative Summary

Table 1 summarizes the architectural features most relevant to physical-access security. The most important differences are not just cipher choices; they are key-release policy, hardware binding, recovery design, and whether ciphertext integrity is available by default.

Table 1: Comparative architecture of four FDE systems. "Default" describes common deployments rather than every possible configuration.

Dimension	BitLocker	FileVault 2	LUKS2	VeraCrypt
Common sector mode	XTS-AES-128/256	Hardware-backed AES-XTS on modern Macs	aes-xts-plain64	XTS with selectable ciphers
Key hierarchy	FVEK wrapped by VMK; VMK protected by key protectors	APFS VEK wrapped by KEK tied to password and hardware UID	Master key unlocked by one or more keyslots	Header keys derived from password/PIM unlock volume keys
Password KDF	Protector-dependent; recovery and TPM modes differ	Password combined with hardware UID/Secure Enclave flow	Argon2 variants or PBKDF2 with stored parameters	PBKDF2-HMAC; PIM controls iteration count
Hardware binding	TPM 1.2/2.0 optional or automatic	Secure Enclave on supported Macs	Optional TPM2 enrollment	None by default
Pre-boot authentication	TPM-only, TPM+PIN, USB, recovery	macOS login/FileVault unlock	Passphrase, keyfile, TPM2, or enterprise tooling	Passphrase/PIM/keyfiles
Integrity protection	Not provided by XTS itself	Not provided by XTS itself	Available with dm-integrity/dm-crypt, not universal	Not provided by XTS itself
Primary strength	Enterprise manageability and platform measurement	Tight hardware-rooted key handling	Flexible, auditable Linux stack	Portability and hidden-volume support
Main physical-access concern	TPM-only bus exposure and boot-policy complexity	Trust concentrated in Apple hardware/firmware chain	Unencrypted boot components unless measured/sealed	Bootloader tampering and passphrase capture

3 Threat Model and Security Goals

Fundamentally, the objective of Full Disk Encryption (FDE) is to ensure the confidentiality of user data while the system is powered off. In this foundational scenario, we assume an attacker who may confiscate the device, extract the storage, capture an image of the encrypted data, and then conduct offline operations, with their success limited only by the cryptographic robustness of the keys and password-based protections. Under these conditions, contemporary FDE solutions offer robust security, provided the system administrator has implemented strong user passwords, established viable recovery mechanisms, and is running up-to-date hardware.

Our attention, however, is directed toward a more capable, physically proximate threat model. This attacker is presumed capable of gaining temporary physical access to the computer, altering boot loaders or other components not encrypted on the disk, connecting monitoring or analysis hardware to system buses, conducting Direct Memory Access (DMA) attacks via a variety of physical ports, leveraging recently powered (and therefore recently active) memory, or even reusing old versions of ciphertext sectors. This adversary is not assumed to possess the user's high-entropy recovery key, nor do we expect the attacker to have access to enough AEGIS-Lock shares to reconstruct it, or to the ability to run arbitrary code within an already-unlocked OS.

We do not pretend that any FDE design is a cure-all for endpoint threats. FDE offers no protection for data once a legitimate user has unlocked the volume and malware is running with a high privilege level. It cannot protect against physical coercion, screen captures, compromise of an online account, user-chosen passwords, or exfiltration at the application level. AEGIS-Lock focuses specifically on those parts of the FDE attack space where we believe there is still room for meaningful improvement: release of keys before boot, tamper-evident sector reads, the ability to recover from a partial factor compromise, and limiting the usable lifetime of the wrapping keys.

4 The Attack Landscape

4.1 Cold Boot Attacks

Cold boot attacks leverage the characteristic that data stored in DRAM is partially recoverable for a brief period after the machine has lost power, particularly if the memory modules are kept cool.

Halderman et al. [1] demonstrated the recovery of the encryption keys from memory, and then used these recovered keys to decrypt the disks protected by typical FDE systems. Several other papers and demonstrations have subsequently updated and improved upon the attack for newer hardware and firmware behavior, but the key lesson from this type of attack remains unchanged: whenever a volume is unlocked, the key or expanded key material is available in some form to the attacker in a recoverable form in live memory.

Mitigations include resetting and overwriting memory, avoiding sleep states that maintain keys, using an IOMMU, using RAM encryption, and architectures such as TRESOR that attempt to keep keys outside of DRAM [6]. While these mitigations are certainly valuable, many of them sit (to some extent) outside of the disk FDE container. A disk-encryption protocol might not reduce all of the data needed to be stored in plaintext for a cold boot attack, but the disk-encryption protocol certainly can't solve cold boot entirely without assistance from the OS, the CPU, or the firmware.

4.2 TPM Bus Sniffing

The ability of the TPM to unlock the disk for us during boot provides the primary usability benefit of TPMs: we can reboot the machine without entering a disk password. This is a double-edged sword, since it implies that at least a portion of the key material, or the secret key, or at the least some part of the wrapping key material that would decrypt the wrapped key material, must be available to the machine from the TPM. If a TPM is an external discrete device (not on-die on the CPU), and it is connected on a bus that an attacker can observe, then the keys can be recovered on that bus. Indeed, Andzakovic demonstrated recovery of BitLocker keys on the BitLocker TPM bus, again using very inexpensive hardware [30]. The same logic also applies to other TPM-only unlock designs, in cases where released material is sufficient to unlock the disk and there is no need for an additional user-specific factor.

In real-world terms, the consequence is fairly obvious: TPM-only is easy to use, but TPM-only is also single factor. Requiring a PIN, passphrase, USB key, network policy, or threshold share alters the economics of an attack because bus sniffing alone no longer supplies all the material needed to unlock the volume.

4.3 Evil Maid Attacks

An evil maid attack alters the boot process to inject a legitimate user into malicious code where they are required to type in a secret. Conventional Linux FDE deployments are vulnerable to evil maid attacks if `/boot` or the initramfs is unencrypted and unmeasured. VeraCrypt and other pre-boot authenticators are also susceptible to these attacks if the bootloader can be replaced. The presence of Secure Boot, TPM PCR binding, and measured boot mitigate the risk of boot process modification, so long as the platform is checked for boot integrity before the passphrase is prompted.

The ordering of checks matters. If a rogue bootloader asks for the passphrase before the platform is checked, it has already won. AEGIS-Lock therefore requires an early abort to be included in the protocol, so that the measured boot integrity is checked before a share is requested from a human.

4.4 DMA Attacks

DMA attacks use peripherals or expasion buses to read or write system memory. Thunderclap showed that IOMMU protections have historically been difficult to deploy correctly and that device-facing interfaces can still expose memory-corruption or memory-disclosure paths [8]. Thunderbolt attacks such as Thunderspy further illustrate the risk of high-speed external buses when physical access is available [9]. For FDE, the danger is that a DMA-capable adversary may extract keys from a running or sleeping machine after the disk has already been unlocked.

As with cold boot, the primary mitigations are platform-level: IOMMU enforcement, kernel DMA protection, port authorization, disabling risky sleep states, and memory encryption. AEGIS-Lock can reduce some key-management exposure, but it cannot make a fully compromised live memory image safe.

4.5 XTS Mode's Integrity Gap

XTS-AES was developed specifically to provide confidentiality for block-level storage media without

havng to store each unit with expansion. NIST states that XTS mode is not designed to authenticate either data or data source [18]. This is not an oversight or a bug, but a design decision with implications. Without data origin authenticity, ciphertext modification may produce corrupted output data rather than a clear authentication failure.

The prototype shows the difference in behavior. One-bit ciphertext modification, when decrypted under the unauthenticated baseline, produces garbage output. The same one-bit ciphertext modification when decrypted under AES-GCM is rejected and the ciphertext is never converted to plaintext. This integriy flaw is somewhat less apparent in many FDE deployments than password cracking or boot process tampering, but it does matter for potential rollback attacks, targeted filesystem modification, and attackers who have the capability to repeatedly image and modify encrypted sectors.

5 AEGIS-Lock: A Threshold-Gated FDE Protocol

We will now propose AEGIS-Lock, which is our research prototype that explores how the existing cryptographic tools can be combined into a stronger FDE key lifecycle. Our design goal with AEGIS-Lock is not to directly replace BitLocker, FileVault, LUKS2, or VeraCrypt directly, rather, we ask what an FDE system would look like if it were to treat physical-acess attacks, sector integrity, and key-release ordering as its first-class protocol requirements.

5.1 Protocol Overview

Figure 1 shown below on page 11 depicts the key hierarchy. AEGIS-Lock separates the volume into effectively three main layers:

1. Threshold unlock layer: We split the 256-bit Master Key (MK) into (k, n) shares, and these are distributed across heterogeneous trust domains. A threshold of shares is required before any wrapping key can be derived.
2. Epoch wrapping layer: The MK derives epoch keys through HKDF. Each epoch key wraps the Volume Encryption Key (VEK). Epoch rotation re-wraps the VEK without re-encrypting disk sectors.
3. Sector encryption layer: We encrypt disk sectors using an AEAD mode with deterministic nonces that are derived from sector number and epoch. Associated data helps to bind every sector with the volume UUID and epoch.

A key design distinction is crucial for correctness. The key-wrapping epoch has control over which epoch key wraps the VEK, and we note that rotating it does not require us to rewrite disk sectors. The sector-write epoch controls the AEAD nonce and AAD that are used in any specific sector write. A production implementation must satisfy that it either persists sector-write epochs/counters, derives them from a trusted copy-on-write layout, or that it enforces a policy which prevents rewriting the same sector under the same sector-write epoch. The Python prototype uses a single epoch argument in the sector functions for simplicity, so we note that the sector tests demonstrate nonce/AAD binding rather than a complete crash-safe metadata system.

5.2 Contribution 1: Commitment-Verified Threshold Shares

Although standard Shamir Secret Sharing [16] does provide threshold secrecy, it does not by itself identify corrupted or malicious shares prior to performing reconstruction. If even one input share is altered, interpolation can produce an incorrect secret, and only a generic decryption failure may be visible to the user. Feldman verifiable secret sharing helps in addressing this problem with public commitments to polynomial coefficients [17].

The prototype is an implementation of a lighter commitment-verified variant that would be suitable for a local FDE header. Each share S_i is accompanied by a HMAC-SHA256 commitment that is keyed by a value derived from the volume UUID. Let us define L_{vss} = "aegis-lock-vss-" to be equal to the fixed domain-separation label:

$$K_u = \text{SHA256}(L_{\text{vss}} \| UUID), \quad (1)$$

$$C_i = \text{HMAC}_{K_u}(S_i). \quad (2)$$

Prior to reconstruction, the boot flow verifies every share that is selected for reconstruction with a constant-time comparison. A corrupted share

will then be rejected by index, which provides the user with actionable recovery information. As an example, the user could be told that share 3 failed verification, rather than just that the whole disk mysteriously failed to unlock.

This construction is best described as being Feldman-inspired rather than being equivalent to a full complete Feldman VSS implementation. If implemented as described, it will give us computational integrity for stored shares, but it will not give us any information-theoretical public verifiability for the original sharing polynomial. For the prototype's threat model, whose goal is to detect share corruption or substitution in a local FDE header before the reconstruction itself, this tradeoff serves to minimize the metadata size and additionally simplifies the implementation. For potential improvements in a production design, such a design could replace HMAC commitments with polynomial commitments instead in the case that it needed publicly verifiable shares across mutually distrustful parties.

5.3 Contribution 2: Epoch-Based Key Wrapping

Often times, traditional key rotation implies that we must do expensive whole-volume re-encryption. AEGIS-Lock demonstrates that this does not always have to be the case, instead separating the lifetime of the VEK from the lifetime of the key that wraps it. Let L_{EK} = "aegis-lock-epoch-key" and L_{wrap} = "aegis-lock-vek-wrap-epoch-":

$$\begin{aligned} \text{EK}_{e_K} &= \text{HKDF}(\text{MK},\ \mathit{salt}\|e_K,\ L_{\text{EK}}), \\ W_{e_K} &= \text{AES-GCM}_{\text{EK}_{e_K}}(\text{VEK},\ L_{\text{wrap}}\|e_K). \end{aligned} \tag{3}$$

The VEK will remain constant throughout the volume's lifetime. Doing a rotation from a key-wrapping epoch e_K to the epoch $e_K + 1$ will derive a fresh epoch key and it will rewrap only the VEK. Hence, there will be no disk sectors that have to be rewritten. In our prototype that we implemented, this operation ran on the order of a couple of hundreds of microseconds (Figure 2).

The security benefit here will be targeted rather than absolute. If an attacker extracts only an epoch wrapping key, then it follows that subsequent wrapping-key rotation invalidates that key for VEK unwraps in the future. If the attacker actually extracts the VEK from the live memory, re-wrapping will not offer any help; that scenario will require additaionl memory-protection techniques, with examples including TRESOR, Intel TME, AMD SME, or careful kernel key handling. The value in the epoch wrapping is hence in narrowing the usefulness that compromised wrapping material has, and not in claiming automatic recovery from full live-key compromise.

Forward-Security Limitation Construction A (the MK-rooted design above) will not provide any forward security in the cryptographic sense. Because each epoch key will be derived independently from the same long-lived MK, if MK is compromised, it will reveal all the past and future epoch keys. This limitation will be mitigated partially by two of our design choices. First, we have MK exist only briefly in RAM during the boot reconstruction window and it should be zeroized right immediately after the EK derivation. Second, the threshold layer helps to make sure that reconstructing MK requires k shares from distinct trust domains. Despite this, an attacker who captures MK during that brief window, for instance, through some type of cold boot attack in the exact time of reconstruction, will obtain all epoch keys.

Alternative: Forward-Secure Ratcheted Wrapping (Construction B)

A stronger alternative replaces the MK-rooted derivation mentioned above with an HKDF chain:

$$K_0 = \text{HKDF}(\text{MK},\ \mathit{salt},\ \texttt{"ratchet-init"}), \tag{4}$$

$$K_{i+1} = \text{HKDF}(K_i,\ \texttt{"ratchet"}\|(i+1)), \tag{5}$$

$$W_i = \text{AES-GCM}_{K_i}(\text{VEK},\ \texttt{"wrap-"}\|i). \tag{6}$$

After deriving K_{i+1}, the system will set K_i to 0. By HKDF's preimage resistance, an attacker who compromises K_T at epoch T can derive $K_{T+1}, K_{T+2}, \ldots$ (forward), but it cannot recover $K_{T-1}, K_{T-2}, \ldots$ (backward). This represents genuine backward secrecy: compromise at a time T will not expose wrapping keys from any of the earlier epochs.

Recovery will still work through the threshold layer. Because K_0 is derived in a deterministic manner from the threshold shares, the recovery procedure can be described through the following

Table 2: Wrapping-key construction tradeoffs. Costs measured in the Python prototype.

Property	A: MK-rooted	B: Ratcheted
Backward secrecy	No; MK derives all	Yes; HKDF preimage resistance
MK liveness in RAM	Brief; boot only	Brief; boot only
Recovery from shares	Direct derivation	Derive K_0; ratchet $i\times$
Recovery cost	Direct HKDF at target epoch	Linear ratchet to target epoch
Crash-safe rotation	Simple; stateless	Requires atomic update
Per-rotation cost	220 μs mean	Same class; machine-dependent
Implementation complexity	Low	Moderate

steps: (1) reconstruct MK from shares, (2) derive $K_0 = \text{HKDF}(\text{MK}, \textit{salt}, \text{"ratchet-init"})$, (3) ratchet forward i times to reach K_i, (4) unwrap VEK. The epoch counter i will be stored within the volume header. Both constructions therefore have the same recovery profile: losing the threshold shares will result in the disk being unrecoverable, and on the other hand, if we preserve them, any state will be able to be rebuilt.

Tradeoff Analysis Table 2 provides a comparison of these two constructions. Construction A is more straightforward in its implementation and does not need any form of sequential state management, rather, it is the default in the prototype. Construction B provides backward secrecy at the cost of an added constraint: the ratchet state (K_i and the epoch counter) must be atomically updated, which requires crash-safe writes to the header. For actual deployment in production, an hybrid approach comes naturally: Construction A as the default mode with Construction B available as an opt-in "high-security" mode for environments where the window of brief MK liveness is interpreted as a risk that cannot be accepted.

5.4 Contribution 3: Epoch-Aware Deterministic Nonces

Random AEAD nonces are common, but storing a random nonce per sector can lead to an unnecessary increase in metadata overhead and can also result in complications in crash consistency. AEGIS-Lock derives its 96-bit AES-GCM nonce deterministically from the sector number and a sector-write epoch e_S:

$$nonce = sector_num \text{ (8B)} \| e_S \text{ (4B)}. \tag{7}$$

Associated authenticated data binds the sector to the volume and write epoch:

$$AAD = sector_num \| volume_UUID \| e_S. \tag{8}$$

This construction can be seen to have three main advantages. First, the same plaintext written to the same sector under different sector-write epochs encrypts differently, limiting temporal traffic analysis when epochs advance. Second, no random nonce bytes must be stored because the nonce is derived from structured metadata. Third, replaying ciphertext from one sector-write epoch into another changes both the nonce context and AAD, causing authentication failure (Figure 3).

The design must still enforce the standard AEAD uniqueness rule: the same key/nonce pair must never be reused for two different plaintexts. AEGIS-Lock satisfies this only if sector writes under a fixed sector-write epoch do not reuse the same sector/epoch nonce after the sector's plaintext changes. A production implementation would therefore need one of three strategies: (1) advance the sector-write epoch before rewriting a sector, (2) store a per-sector write counter and include it in the nonce or AAD, or (3) use a copy-on-write layout that always writes changed data to a fresh logical address. The prototype focuses on demonstrating the mechanism and replay binding; crash-safe sector-epoch metadata is intentionally left as an implementation requirement rather than assumed away.

Nonce Uniqueness Invariant The AEGIS-Lock nonce uniqueness invariant can be stated precisely. Let $\mathcal{E}(k, n, pt)$ denote the encryption of plaintext pt under key k and nonce n. The safety requirement is:

$$\forall (s, e_S), \quad \#\{pt : \mathcal{E}(VEK,\ s\|e_S,\ pt) \text{ is ever used}\} \le 1. \tag{9}$$

Under this constraint, the nonce space of 2^{64} sector addresses $\times$ 2^{32} sector-write epochs provides a total nonce capacity of 2^{96} unique encryptions before any structural collision. For a 1 TB volume with 2^{28} 4 KiB sectors and hourly global sector-epoch

advancement, exhausting this nonce space would take far longer than the practical lifetime of any storage device. Strategies (1) and (3) integrate naturally with log-structured or copy-on-write filesystems such as ZFS, Btrfs, or APFS. Strategy (2) is more compatible with conventional in-place-update filesystems such as ext4 or NTFS, but it requires persistent per-sector metadata. Thus the 16-byte-per-sector overhead reported below is the cryptographic tag overhead; additional metadata depends on the chosen rewrite strategy.

5.5 Contribution 4: TPM-First Ordered Boot

AEGIS-Lock defines the boot process as an ordered protocol rather than a loose set of checks. The critical rule is that platform measurement verification must happen before any user-entered share is requested. If the TPM PCR policy fails, the system aborts before displaying a passphrase prompt. This prevents a compromised bootloader from collecting the passphrase after the platform has already become untrusted.

The recommended measured set includes firmware, boot manager, Secure Boot policy, initramfs, and FDE access-control state. On Linux-style deployments, measuring the initramfs is particularly important because an attacker who can modify early userspace can wait until after key release and then exfiltrate secrets.

5.6 Contribution 5: Cross-Epoch Replay Prevention

While authenticated sectors effectively guard against bit-flipping, storage systems must also address rollback attacks. AEGIS-Lock mitigates this by binding the sector-write epoch to both nonce generation and the associated authenticated data (AAD). Consequently, ciphertext and tag pairs produced during sector-write epoch e_S will be rejected if transferred to a sector where the trusted sector metadata indicates a different epoch, e'_S.

However, AEGIS-Lock alone does not stop an adversary from reverting the whole disk image (and associated metadata) to some consistent snapshot. Full-device rollback resilience is still necessary, and typically depends on monotonic counters, TPM NV indices, server-side freshness assertions, or other trusted anti-rollback anchors. What AEGIS-Lock asserts is that cross-epoch sector substitution is not sufficient to pass authentication under protection of the expected sector epoch.

Algorithm 1 AEGIS-Lock Ordered Boot Protocol

Require: Header H, threshold k, epoch e, PCR policy $\mathcal{P}$
Ensure: VEK or abort diagnostic

1: **Stage 1: Platform Verification**
2: $p \leftarrow$ TPM2PCRRead($\mathcal{P}$)
3: **if** $p \neq H.expectedPCRs$ **then**
4:     **return ABORT**: PCR mismatch; no prompt
5: **end if**
6: **Stage 2: Share Collection**
7: $\mathcal{S} \leftarrow$ CollectShares(TPM, P, U, N)
    where P=passphrase, U=USB, N=network share
8: **if** $|\mathcal{S}| < k$ **then**
9:     **return ABORT**: insufficient shares
10: **end if**
11: **Stage 3: Share Verification**
12: **for** $(x_i, S_i) \in \mathcal{S}$ **do**
13:     **if** $\text{HMAC}_{K_u}(S_i) \neq H.commit[x_i]$ **then**
14:         **return ABORT**: share x_i failed
15:     **end if**
16: **end for**
17: **Stage 4: Key Reconstruction**
18: MK $\leftarrow$ Interpolate($\mathcal{S}[1{:}k], 0$)
19: **Stage 5: EK Derivation and VEK Unwrap**
20: $\text{EK}_e \leftarrow \text{HKDF}(\text{MK}, e \| salt,$ `"epoch-key"`$)$
21: $\text{VEK} \leftarrow \text{AES-GCM}^{-1}_{\text{EK}_e}(H.wrappedVEK)$
22: **if** authentication fails **then**
23:     **return ABORT**: VEK unwrap failed
24: **end if**
25: **Stage 6: Test Decryption**
26: Verify a known test sector with VEK
27: **return** VEK

5.7 Proposed Share Configuration

The recommended configuration is a $(3, 5)$ threshold:

1. S_1: TPM-sealed share bound to measured boot state.
2. S_2: passphrase-derived share using a memory-hard KDF such as Argon2id.
3. S_3: USB security key, FIDO2 token, or equivalent possession factor.
4. S_4: network-bound share through enterprise infrastructure such as Tang/Clevis.
5. S_5: recovery escrow held offline by the user or administrator.

Any three shares suffice to reconstruct the MK.

This threshold distribution is intentionally diverse: a TPM bus sniff is sufficient to leak at most S_1; phishing or shoulder surfing may reveal at most S_2; and losing a USB token results in at most loss of S_3. If any one or two of these share types is unavailable, the system may be restored with the other two types, but no one share type is sufficient to gain access to the disk.

The next improvement over AEGIS-Lock is to use the threshold shares to compute a threshold pseudorandom function (PRF) rather than to recombine the shares. Each share owner can instead run a partial evaluation of the PRF over the current volume UUID and sector epoch. These partial evaluations would then be combined by the boot environment to generate the epoch key, which is then used to unwrap the VEK.

In this variant, the MK never appears in normal memory at any time; instead, the PRF value appears in normal memory during the boot phase. This has two beneficial effects: first, it still retains the threshold property of the underlying scheme, and second, it eliminates the risk that memory exposure of the master key (as discussed above) during boot would allow reconstruction of all other master keys.

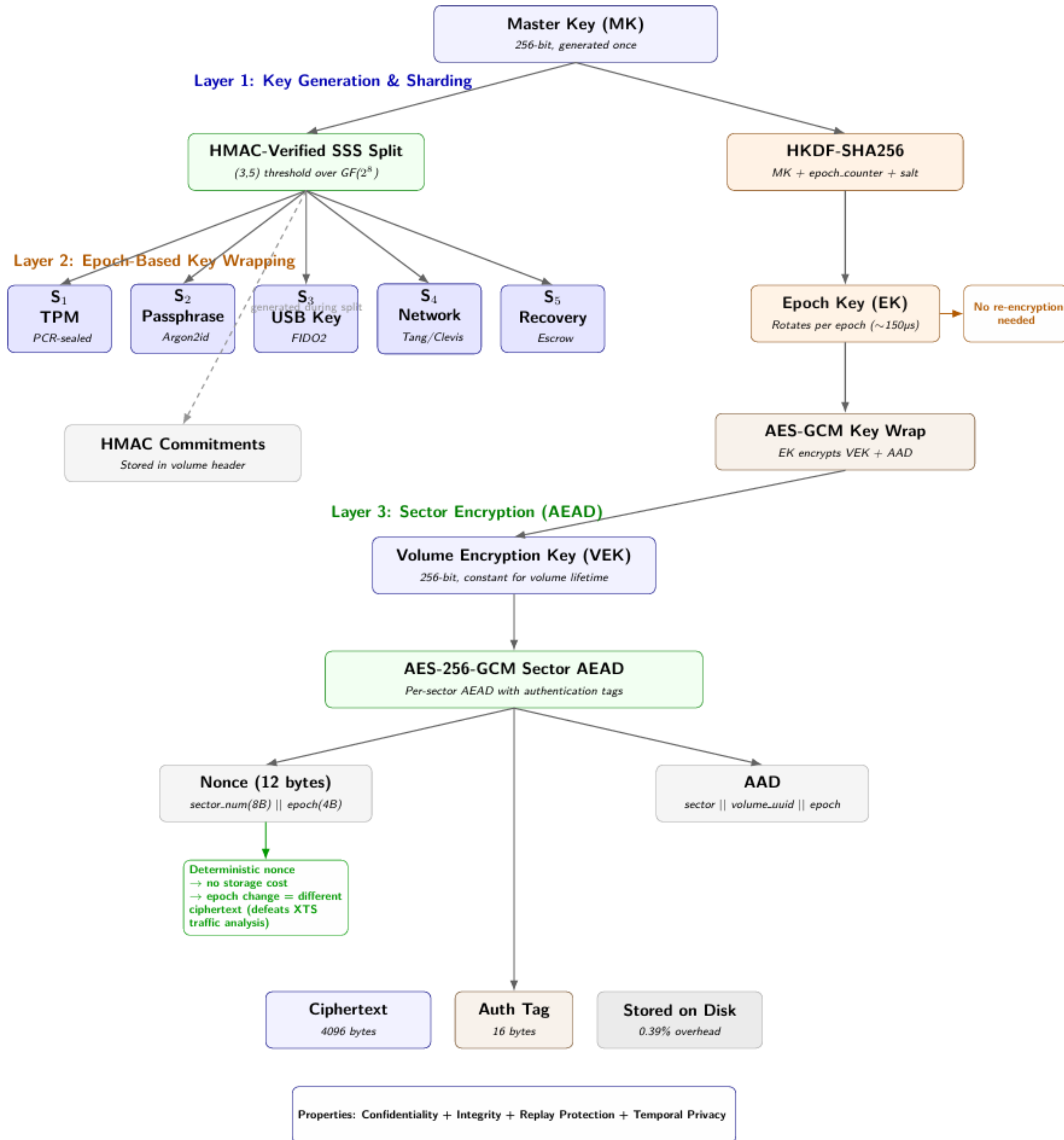


Figure 1: Key hierarchy for our proposed scheme. The Master Key (MK) gets sharded into five shares across: TPM, passphrase, USB token, network unlock, and recovery escrow. Epoch Keys that are derived through the HKDF are used to wrap the constant VEK. Sector encryption uses AEAD with epoch-aware deterministic nonces.

Epoch rotation: VEK remains constant; only wrapping changes (∼150μs)

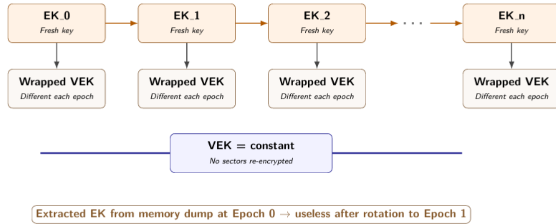
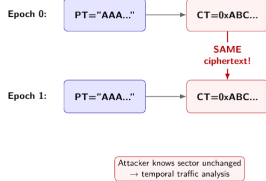


Figure 2: Epoch rotation re-wraps the VEK under a fresh Epoch Key in roughly a few hundred microseconds in the Python prototype. The VEK remains constant, so no sectors are re-encrypted.

XTS-AES (current systems)



AEGIS-Lock (epoch-aware nonce)

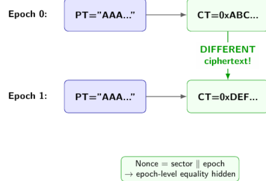


Figure 3: XTS produces the same ciphertext for the same plaintext at the same sector, which can reveal temporal patterns. AEGIS-Lock's epoch-aware nonces change ciphertext across epochs while keeping nonce storage at zero bytes per sector.

AEGIS-Lock: Ordered Boot Protocol

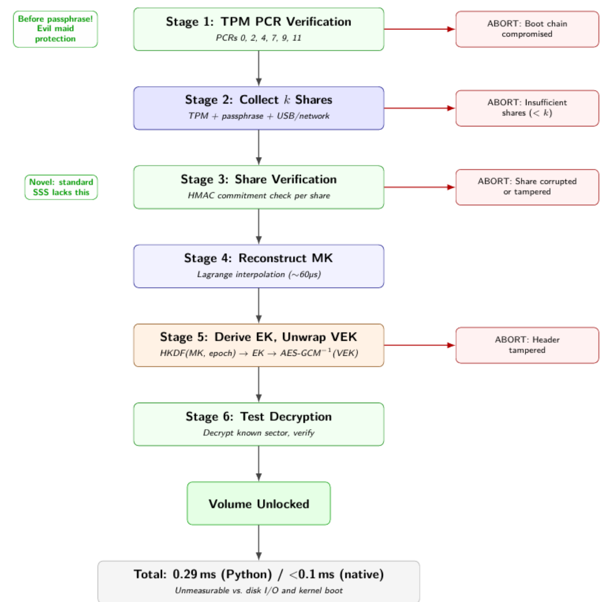


Figure 4: AEGIS-Lock boot protocol. Each stage aborts on failure. The TPM/PCR check runs before the passphrase prompt, so a boot-chain mismatch prevents evil-maid passphrase capture. Share verification also occurs before reconstruction, allowing corrupted shares to be identified directly.

6 Implementation and Evaluation

We built the prototype in Python using PyCryptodome. We implement Shamir Sharing over the field GF(2^8) using irreducible polynomial $x^8 + x^4 + x^3 + x^2 + 1$ (0x11D). HKDF (extract-and-expand) was implemented per the spec in RFC 5869 [26]. AES-GCM is used for both VEK wrapping and sector-level AEAD experiments. Benchmarks were collected in a containerized Linux environment, so absolute throughput should be interpreted as prototype evidence rather than native-kernel performance.

6.1 Threshold Sharing Performance

In our SSS benchmarks we tested 128-bit and 256-bit keys, and thresholds $k \in 2, 3, 5$ for $n \in 3, 5, 7, 10$ sharing schemes (Figure 5); we repeated each test 50 times to average the overheads. The checked-in run's $(k, n) = (3, 5)$, 256-bit AES key has split and reconstruction times of 479 μs and 101 μs, respectively. Even the largest tested configuration of $(k, n) = (5, 10)$, 256-bit, is only around 1.1 ms and 254 μs.

6.2 Sector Encryption: XTS Baseline vs. AEAD

We compare AES-CBC as a bacic, unauthenticated baseline against AES-GCM over 4,KiB sectors (Figure 6). Python throughput in the checked-in run is roughly 200-217 MB/s for the baseline, and about 34-38 MB/s for GCM over 4 KiB sectors. We caution not to read the 4× cost directly as the expected production cost; a kernel or storage-stack implementation that utilizes AES-NI and optimized I/O scheduling would likely be far faster. Instead, Pyton throughput is not the most interesting result; the storage and correctness tradeoff is: AEAD adds authentication and tamper rejection, and in return, one must store the tags and manage nonce uniqueness.

For 4,KiB sectors, a 16-byte auth tag introduces a $16/4096 = 0.390625\%$ storage overhead. If a random-noce scheme (e.g. generic GCM) were used, an additional 12-byte nonce would be stored for a total overhead of $28/4096 = 0.68\%$. As AEGIS-Lock's deterministic nonce construction avoids any extra nonce storage, the crypto-tag cost amounts to 16 bytes per sector (Figure 7). (In a real system, one may still want to store sector-epoch or write counter metadata, depending on the particular rewrite strategy.)

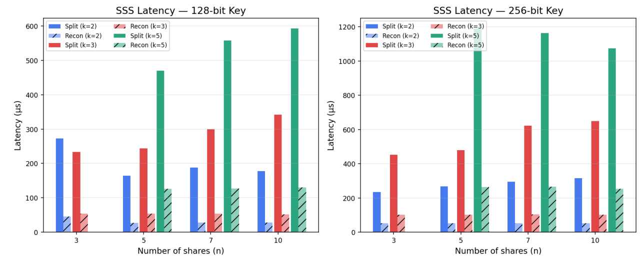


Figure 5: Secret-sharing latency. Split time scales with n, k, and key length; reconstruction depends primarily on k. The recommended 256-bit $(3, 5)$ configuration completes in roughly 0.48 ms for splitting and 0.10 ms for reconstruction in the checked-in run.

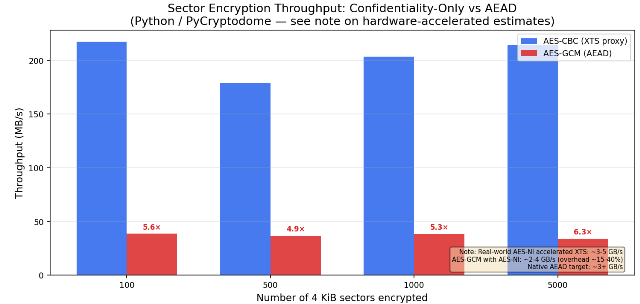


Figure 6: Prototype sector-encryption throughput in Python. The GCM penalty is inflated by interpreter overhead and should be treated as a conservative prototype baseline, not a native implementation forecast.

6.3 Key Derivation Cost

Table 3 and Figure 8 show PBKDF2-HMAC-SHA256 timings for iteration counts representative of legacy and modern FDE systems. The main leson is qualitative: low iteration counts are cheap enough to invite large-scale guessing, while modern settings and memory-hard KDFs impose a much higher cost per trial. For password-derived shares, AEGIS-Lock should use Argon2id or an equivalent memory-hard KDF rather than relying on PBKDF2 alone.

Table 4 and Figure 9 benchmark Argon2id at configurations recommended for password hashing. In the checked-in run, the lightest Argon2id configuration ($t = 1$, 64 MB, $p = 4$) costs about 60 ms per trial, while the 64 MB four-pass configuration costs about 139 ms. The heaviest tested configuration

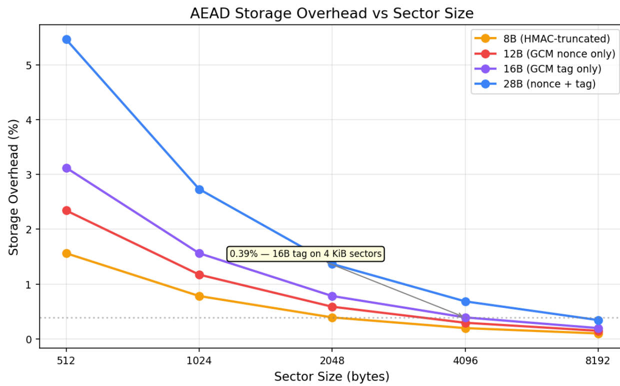


Figure 7: AEAD storage overhead by sector and metadata size. AEGIS-Lock targets the 16-byte-tag line because deterministic nonces need not be stored. For 4 KiB sectors, that overhead is 0.39%.

Table 3: PBKDF2 key-derivation time in the Python benchmark.

Configuration	Iterations	Time (ms)
Legacy low-cost example	1,000	0.6
200k-iteration example	200,000	111.1
500k-iteration example	500,000	280.1
High-cost example	1,000,000	558.1

Table 4: Argon2id key-derivation time. Memory hardness resists GPU/ASIC parallelism.

Configuration	Memory	Passes (t)	Time (ms)
OWASP minimum	64 MB	1	60.5
Moderate	64 MB	2	88.5
Higher security	64 MB	3	114.7
Time-heavy	64 MB	4	139.1
Memory-heavy	256 MB	2	350.0

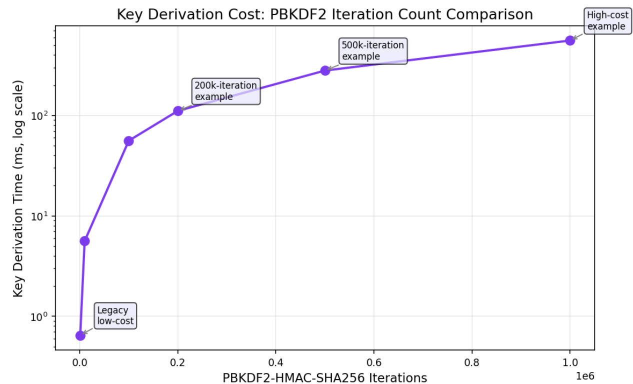


Figure 8: PBKDF2 derivation cost across representative iteration counts. The log scale emphasizes the large gap between legacy and modern settings.

($t = 2$, 256 MB, $p = 4$) costs about 350 ms per trial. These values are machine-dependent, but they show that memory-hard evaluation remains feasible at boot while resisting GPU and ASIC parallelism better than PBKDF2 alone. For AEGIS-Lock's passphrase-derived share S_2, Argon2id at $t \geq 2$ and $m \geq 64$ MB provides the recommended minimum resistance.

6.4 Boot Protocol Timing

Table 5 and Figure 10 report representative per-stage timing for the AEGIS-Lock boot simulation over 200 local trials.. The simulated TPM check is not a real hardware TPM transaction, so the absolute TPM row should not be interpreted as production timing. The thresshold, verification, reconstruction, HKDF, unwrap, and test-decryption steps together are still small relative to ordinary boot costs such as firmware iniitialization, kernel loading, and disk I/O.

6.5 Epoch Rotation Performance

The checked-in benchmark reports MK-rooted epoch rotation over 100 trials. Construction A averages 220 μs per rotation: it derives a fresh epoch key from MK and re-wraps only the VEK, so no data sectors are rewritten. After rotation, the VEK remains unchanged and the volume still decrypts correctly.

The prototype also implements Construction B, the ratcheted wrapping variant described above, in `aegis_lock.py`. Construction B has the same operation class during normal rotation-one HKDF step and one AES-GCM VEK wrap-but recovery is linear in the epoch count because the system must rachet forward from K_0 to the target epoch. For that reason, the paper treats Construction B as a security/usability tradeoff rather than claiming a precise persisted benchmark for it in `simulation_results.json`.

6.6 Tamper Detection and Replay Demonstrations

The tamper test flips one bit in the ciphertext. Under the unauthenticated scheme, the decryption succeeds with corrupted plainext; 17 plaintext bytes change in an unnoticeable way. Under AES-GCM, the same modification raises an auth error and no plaintext is released. The replay test encrypts

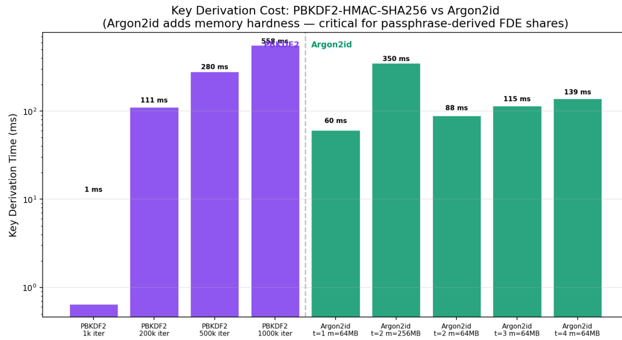


Figure 9: Key derivation cost: PBKDF2-HMAC-SHA256 vs Argon2id. Argon2id's memory hardness makes it resistant to GPU and ASIC parallelism, which is critical for passphrase-derived FDE shares.

Table 5: AEGIS-Lock boot protocol timing (Python, 200 local trials).

Stage	**Mean (μs)**
1. TPM PCR check (simulated)	0.2
2. Share collection	0.8
3. Share commitment verification	18.7
4. MK reconstruction	107.8
5. EK derivation + VEK unwrap	142.8
6. Test decryption	193.7
Total	**464.0**

the same plaintext under epochs 0 and 1 to sector 42. The ciphertexts are different, and attempting to decrypt under epoch-1 ciphertext using epoch-1 parameters fails. Together these experiments illustrate the difference between confidentiality-only and authenticated disk encryption.

6.7 Full-Disk Performance Extrapolation

Table 6 gives a hardware-level extrapolation for whole-volume encryption time using a few representative AES-NI throughputs. The numbers are given with an intentionally optimistic interpretation compared to the Python prototype; however, these times are more representative of the expected scale given a native implementation where AES instructions and storage throughput become the dominant factors.

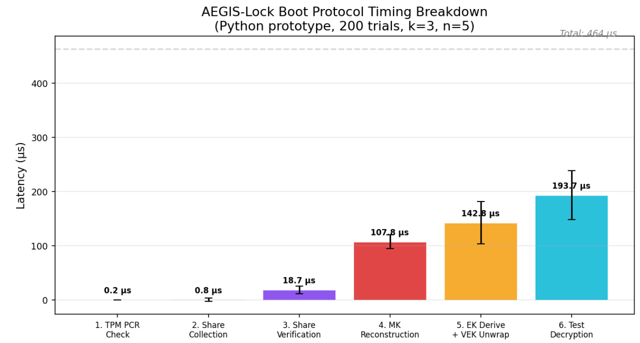


Figure 10: AEGIS-Lock boot protocol timing breakdown. Reconstruction and VEK unwrap dominate; the total is well under 1 ms in the Python prototype and would be smaller in a native implementation.

Table 6: Full-disk encryption time extrapolation under representative hardware-accelerated throughput.

Size	**XTS (4 GB/s)**	**GCM (3 GB/s)**	**AEAD target (3.5 GB/s)**
100 GB	0.4 min	0.6 min	0.5 min
500 GB	2.1 min	2.8 min	2.4 min
1 TB	4.3 min	5.7 min	4.9 min

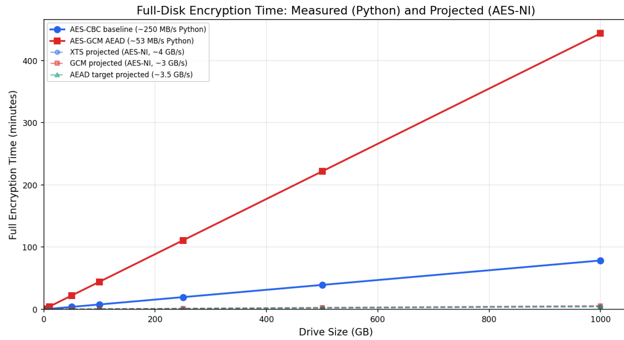


Figure 11: Full-disk encryption time: measured Python extrapolation and hardware-accelerated projection. The projected lines are the more relevant comparison for a native disk-encryption layer.

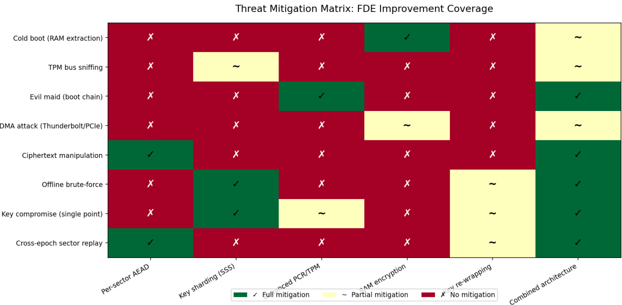


Figure 12: Threat mitigation heatmap. The combined architecture covers more attack classes than any single mechanism, but DMA and cold boot still depend on platform-level memory protections.

7 Threat Mitigation Analysis

7.1 Mitigation Matrix

Table 8 and Figure 12 present the coverage of AEGIS-Lock across a range of eight attacks. As you can see, the table separates mechanisms since each improvement alone is not sufficient. AEAD protects against tampering and sector replay, threshold sharing protects against the loss of a single factor of protection, PCR-first boot protects against evil-maid capture, and RAM encryption is needed for cold boot and DMA resistance.

7.2 Security Properties Comparison

Table 7 compares the individual security properties of several existing FDE systems and AEGIS-Lock. The table focuses on these structural properties as opposed to the resistance to specific attacks, serving as a complement to the mitigation matrix presented above.

7.3 New Attack Surfaces and Limitations

While AEGIS-Lock does overcome some of the weaknesses of currently deployed FDE systems, it unfortunately introduces

First, threshold sharing introduces operational risk to users and administrators. Users and administrators must now protect multiple shares, rotate them periodically, and recover shares if they are lost or compromised. A poor implementation of a recovery mechanism may become the weakest link, while an overly restrictive policy may lock a legitimate user out.

Second, AEAD sector encryption relies on careful nonce management. We can easily avoid the noce storage overhead by simply using a deterministic sector/epoch pair as our nonce in a production environment, but care must be taken to ensure that the same sector is not written under the same epoch with a different plaintext. Per-sector counters, write-once advances to epochs, or a COW filesystem can help solve this issue, but the issue cannot be ignored. Additionally, the entire device cannot be rolled back using our sector-level AEAD unless epoch metadata itself can be bound to a monotonic TPM counter or some other externally-verified freshness source.

Third, AEGIS-Lock requies a larger trusted computing base. A full implementation relies on a working kernel storage stack, TPM tooling, secure boot, a threshold share implementation, a KDF implementation, an AEAD library, and possibly network-unlock infrastructure. All components must fail securely. Our Python prototype consists of ≈830 lines of code for the AEGIS-Lock core protocol and demonstration harness (which includes Shamir sharing, HKDF, epoch wrapping, sector AEAD encryption, a boot simulation, and output of the demo) and 780 lines of code for the benchmark suite. A production C implementation is likely to be somewhat smaller, but must also provide dm-crypt hooks, TPM2 stack hooks, Clevis/Tang network share hooks, and an implementation of Argon2id for passphrase hashing. The implementation surface of AEGIS-Lock is significantly larger than the cryptographic mechanisms

Table 7: Security property comparison across FDE systems. Properties reflect default or common deployments.

Property	BitLocker	FileVault 2	LUKS2	VeraCrypt	AEGIS-Lock
Sector authentication	No (XTS)	No (XTS)	Optional (dm-integrity)	No (XTS)	Yes (AES-GCM per-sector)
Unlock factors required	1 (TPM) or 2 (TPM+PIN)	1 (password)	1 (passphrase or keyslot)	1 (passphrase)	k-of-n threshold (default 3-of-5)
Share corruption detection	N/A	N/A	N/A	N/A	HMAC commitment per share
Key rotation without re-encryption	No	No	Possible (rekey keyslot)	No	Yes (epoch-based VEK re-wrap)
Cross-epoch replay prevention	No	No	No	No	Yes (epoch in nonce and AAD)
Boot-chain verification before credential entry	Depends on PCR policy	Firmware-enforced	Depends on setup	No	Protocol requirement (Stage 1)
Memory-hard KDF	No (PBKDF2 in recovery)	Hardware-bound	Yes (Argon2id)	No (PBKDF2)	Yes (Argon2id for passphrase share)
Nonce storage per sector	0 bytes (XTS tweak)	0 bytes (XTS tweak)	Optional	0 bytes (XTS tweak)	0 bytes for nonce; epoch/counter metadata depends on rewrite policy

Table 8: Threat mitigation matrix. ✓ = substantially mitigated, ∼ = partially mitigated or dependent on external controls, × = not addressed by that mechanism.

Attack	AEAD	Threshold	PCR-first	RAM enc.	Re-wrap	AEGIS
Cold boot	×	×	×	✓	×	∼
TPM bus sniffing	×	∼	×	×	×	∼
Evil maid	×	×	✓	×	×	✓
DMA attack	×	×	×	∼	×	∼
Ciphertext tamper	✓	×	×	×	×	✓
Offline brute-force	×	✓	×	×	∼	✓
Single-factor	×	✓	∼	×	∼	✓
Cross-epoch replay	✓	×	×	×	∼	✓

we use: cryptsetup (LUKS 2.0's main command) is approximately 25,000 lines of C and the threshold-share, epoch-wrapping, AEAD-sector-encryption, and ordered-boot mechanisms of our design contribute an additional 3,000-5,000 lines to a LUKS 2.0 system.

Last, our design does not eliminate all atacks against a live system. If an adversary can run malware on a computer after the disk has been unlocked, or if they have a DMA interface that lets them get an unrestricted view of live memory, they will have access to the data in its cleartext, in which case FDE will not protect it. AEGIS-Lock mitigates the pre-boot and at-rest attack space, but it is not an alterative to the hardening of an endpoint device.

8 Related Work and Existing Prototypes

The closest deployed building blocks are already in existence in different pieces. Linux dm-crypt gives us block-device encryption, and dm-integrity can be combined with dm-crypt to provide authenticated disk encryption semantics [10, 21]. LUKS2 allows us to have flexible metadata, keyslots, and KDF parameters [20]. BitLocker depicts how TPM measurement and enterprise recovery can be integrated into a mainstream OS. FileVault is a vivid demonstration of the benefit of connecting key management with a hardware security processor [24], and furthermore, VeraCrypt is an illustration in the value of portability and unlock policy controlled by the user [25].

We will now spend some time drawing out explicitly the dm-integrity comparison. Brož et al. [10] proved that dm-crypt, when done in combination with dm-integrity, has the ability to provide authen-

ticated encryption at the block layer, and the kernel documentation describes how to provision this [21]. AEGIS-Lock's AEAD sector layer gives us an analogous integrity guarantee. The key differences are not in the sector-level mechanism but actually in the protocol that surrounds it. AEGIS-Lock adds threshold-gated key release, epoch-based wrapping, and ordered boot verification, and none of these factors are ever addressed by dm-integrity alone. In practice, dm-integrity is available today in Linux kernels that are in production, while AEGIS-Lock remains moreso a research prototype at the moment. A plausible production path would likely start by integrating AEGIS-Lock's key-lifecycle features into LUKS2, utilizing the dm-integrity for sector-level AEAD instead of doing a reimplementation of it.

Meijer and van Gastel [12] demonstrated that hardware Self-Encrypting Drives (SEDs) contain implementation weaknesses capable of defeating encryption altogether. This highlights the advantage of software Full-Disk Encryption (FDE), where both the encryption algorithm and the key-management logic are openly auditable. Benadjila et al. [11] provide a broader overview of storage confidentiality and integrity, encompassing both block-device and filesystem-level protections.

Academic work has formalized security notions for disk encryption and evaluated the gap between theory and deployed FDE systems [3–5]. Cold boot, DMA, and TPM-bus attacks demonstrate that many failures occur outside the block-cipher abstraction [1,8,30]. AEGIS-Lock draws from these prior studies to outline a single, cohesive protocol that integrates threshold-gated key release, explicit attestation for share validation, ordered boot verification, authenticated sectors, and an efficient rotation of wrapped keys.

9 Conclusion

Even though FDE is effective, its guarantees are often narrower and weaker in real world scenarios. For example, while XTS-AES protects confidentiality, it does not provide integrity. TPM-only unlock is convenient but can become a single-factor hardware-release path. Bootloaders images can turn into passphrase-capture points if the system prompts prior to verifying its platform state. Live memory continues to be a difficult boundary for every design that has to decrypt data in its use.

AEGIS-Lock mitigates some of these risks by introducing a novel FDE protocol featuring threshold-gated key release. Shares that are verified against their cryptographic commitments prevent reconstruction of a wrong key and reduce the exposure of the scheme to single-factor attacks. The wrapping strategy permits frequent key-rotation within the storage scheme without requiring a complete volume re-encryption. An "Authenticated-Encryption-After-Authentication" (AEAD) layer provides sector integrity, while the AAD field incorporates sector-write-epoch information to thwart sector replacement across epochs, provided the wrapping-epoch metadata itself remains intact. The TPM-first boot ordering (Algorithm 1) ensures that a passphrase is requested *only* if the measured boot state is correct; otherwise, the boot sequence is aborted prior to any passphrase prompt. Our Python prototype indicates that the added cost for this enhanced key-management scheme is negligible in practice compared to standard FDE operations (e.g., disk I/O, cryptographic primitives, or key derivations): share reconstruction for a standard 3-of-5 configuration requires merely 0.10 ms, the ordered boot sequence completes in 0.46 ms within the local, checked-in implementation, and MK-rooted wrapping-epoch rotation operates in 250 s; the Argon2id benchmarks confirm that memory-hard key derivations for passphrase-derived shares are practical within a typical boot sequence.

The most important future work in our view would be to see how to get our work into real world production. A production-quality version would need additional components, including things like integration with LUKS2, native AES acceleration, crash-safe metadata updates, and more detailed formal analysis into the boot and key-lifecycle protocol. Even with these seemingly open questions, AEGIS-Lock provides a powerful demonstration that stronger FDE guarantees are technically plausible, and we can do it without abandoning the practical design principle that makes FDE deployable: encrypt the sectors once, and then follow it up by managing keys around them carefully.

References

[1] J. A. Halderman et al., "Lest we remember: Cold boot attacks on encryption keys," in *USENIX Security*, 2008.

[2] C. Fruhwirth, "New methods in hard disk encryption," Master's thesis, Vienna University of Technology, 2005.

[3] L. Khati, N. Mouha, and D. Vergnaud, "Full disk encryption: Bridging theory and practice," in *CT-RSA*, LNCS 10159, 2017.

[4] K. Gjøsteen, "Security notions for disk encryption," in *ESORICS*, LNCS 3679, 2005.

[5] T. Müller and F. Freiling, "A systematic assessment of the security of full disk encryption," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 5, 2015.

[6] T. Müller, F. Freiling, and A. Dewald, "TRESOR runs encryption securely outside RAM," in *USENIX Security*, 2011.

[7] E.-O. Blass and W. Robertson, "TRESOR-HUNT: Attacking CPU-bound encryption," in *ACSAC*, ACM, 2012.

[8] A. T. Markettos et al., "Thunderclap: Exploring vulnerabilities in operating system IOMMU protection," in *NDSS*, 2019.

[9] B. Ruytenberg, "Breaking Thunderbolt protocol security," Eindhoven University of Technology, 2020.

[10] M. Brož et al., "Practical cryptographic data integrity protection with full disk encryption," in *IFIP SEC*, 2018.

[11] R. Benadjila, P. Music, and D. Vergnaud, "Secure storage-confidentiality and authentication," *Computer Science Review*, vol. 44, 2022.

[12] C. Meijer and B. van Gastel, "Self-encrypting deception: Weaknesses in the encryption of solid state drives," in *IEEE Symposium on Security and Privacy*, 2019.

[13] P. Crowley and E. Biggers, "Adiantum: Length-preserving encryption for entry-level processors," *IACR Transactions on Symmetric Cryptology*, 2018(4).

[14] S. Bossi and A. Visconti, "What users should know about full disk encryption based on LUKS," in *CANS*, LNCS 9476, 2015.

[15] O. Choudary, "Infiltrate the vault: Security analysis and decryption of Lion full disk encryption," IACR ePrint 2012/374.

[16] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, 1979.

[17] P. Feldman, "A practical scheme for non-interactive verifiable secret sharing," in *FOCS*, 1987.

[18] M. Dworkin, "SP 800-38E: The XTS-AES mode for confidentiality on storage devices," National Institute of Standards and Technology, 2010.

[19] IEEE Std 1619-2007, "Standard for cryptographic protection of data on block-oriented storage devices."

[20] M. Brož, "LUKS2 on-disk format specification, version 1.1.x," cryptsetup project, 2025.

[21] Linux kernel documentation, "dm-integrity," 2025. Linux kernel documentation.

[22] Microsoft Learn, "Windows Server shows PCR7 configuration as 'Binding Not Possible'," 2026. `https://learn.microsoft.com/en-us/troubleshoot/windows-server/setup-upgrade-and-drivers/pcr7-configuration-binding-not-possible`

[23] Microsoft Learn, "ProtectKeyWithTPM method of the Win32_EncryptableVolume class," 2021. Microsoft Learn.

[24] Apple Platform Security, "Volume encryption with FileVault in macOS," 2026. Apple Platform Security.

[25] VeraCrypt Documentation, "Personal Iterations Multiplier (PIM)." VeraCrypt documentation.

[26] H. Krawczyk and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)," RFC 5869, 2010.

[27] iSEC Partners/NCC Group, "Open Crypto Audit Project TrueCrypt security assessment, Phase I," 2014.

[28] NCC Group, "Open Crypto Audit Project TrueCrypt cryptographic review, Phase II," 2015.

[29] Quarkslab/OSTIF, "Security assessment of VeraCrypt," 2016.

[30] D. Andzakovic, "Extracting BitLocker keys from a TPM," Pulse Security, 2019.

[31] D. Moghimi et al., "TPM-FAIL: TPM meets timing and lattice attacks," in *USENIX Security*, 2020.

[32] C. Cremers, A. Dax, and A. Naska, "Formal analysis of SPDM," in *USENIX Security*, 2023.

A Simulation Code Summary

Table 9 summarizes the submitted Python modules. All correctness assertions pass in the provided prototype. Install dependencies with `pi pinstall-rrequirements.txt`. Then run `python3aegis_lock.py` to reproduce the protocol demonstration and `python3fde_simulations.py` to regenerate the benchmark outputs, including Argon2id and boot timing data. The full benchmark can take several minutes because Argon2id deliberately uses memory-hard settings. Benchmark times vary by machine, so the checked-in `simulation_results.json` is the source for the paper's plotted microbenchmarks.

Table 9: Provided simulation and figure-generation code.

Module	Contents
`aegis_lock.py`	End-to-end AEGIS-Lock prototype: threshold share generation and verification, HKDF epoch-key derivation, VEK wrapping and rotation, AEAD sector encryption/decryption, boot-protocol simulation, and replay/tamper demonstrations.
`fde_simulations.py`	Benchmark suite for secret-sharing latency, unauthenticated vs. AEAD sector encryption, PBKDF2 cost, Argon2id cost, key re-wrapping, tamper detection, boot protocol timing, epoch rotation timing, full-disk extrapolation, and the threat-mitigation matrix.
`generate_figures.py`	Generates the evaluation figures used in the paper, including the boot timing breakdown and KDF comparison charts.
`generate_aegis_figures.py`	Generates the AEGIS-Lock architecture, boot-ordering, nonce-comparicon, and epoch-rotation figures.